Functional Programming

Overview

- Functional vs. imperative programming
- Fundamental concepts
- Evaluation strategies
- Pattern matching
- Higher order functions
- Lazy lists

References

A Bit of History

- Lambda calculus (Church, 1932-33):
  - Formal model of computation
- Lisp (McCarthy, 1960):
  - Symbolic computations with lists
- APL (Iverson, 1962):
  - Algebraic programming with arrays
- ISWIM (Landin, 1966):
  - Let and where clauses
  - Equational reasoning; birth or “pure” functional programming
- Miranda (Turner, 1985)
  - Lazy evaluation
- Haskell (Hudak, Wadler, et al., 1988):
  - “Grand unification” of functional languages
Programs in pure functional languages have no explicit state.

Programs are constructed entirely by composing expressions (functions).

The main concept in functional programming is the application of a function \( f \) to an argument \( e \), written \( fe \).
Side Effects

- Consider the expression $f + g$ and suppose we wish to calculate the two sub-expressions $f$ and $g$ and then add them.

- An optimizing compiler might decide to calculate $f$ first and then $g$ or the other way round or might, if possible, try to calculate both expressions in parallel.

- If the calculation of one of them has the effect of changing the value of the other (via a side effect), then we would be in trouble, because the result of $f + g$ would depend on the order of the evaluation of the sub-expressions $f$ and $g$.

- Such a language is not “referentially transparent”. In general, every programming language that provides imperative programming abstractions belongs to this class of languages.
A function has the property of “referential transparency” if its value depends only on the values of its parameters.

Does \( f(x) + f(x) \) equal \( 2 \times f(x) \)? In C? In Haskell?

Referential transparency means that “equals can be replaced by equals”.

Referential transparency means that a well-formed sub-expression can be replaced by another with the same value without effecting the value of the whole expression.

Pure functional languages, like Haskell, have this property and therefore, in these languages functions yield the same result no matter how often they are called.
Pure Functional Programming

- What is a program?
  - A program (computation) is a transformation from input data to output data.

- Imperative programming: program = algorithm + data

- Functional programming: program = function · function

- Key features of pure functional languages:
  - All programs and procedures are functions.
  - There are no variables or assignments – only input parameters.
  - There are no loops – only recursive functions.
  - The value of a function depends only on the values of its parameters.
  - Functions are first-class values.
Haskell is a general purpose, purely functional programming language incorporating many recent innovations in programming language design. Haskell provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module system, a monadic I/O system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers. Haskell is both the culmination and solidification of many years of research on lazy functional languages.

- The Haskell 98 report.
Haskell Program Structure

- At the topmost level a Haskell program is a set of modules (scripts) that provide a way to control namespaces and to re-use software in large programs.

- The top level of a module consists of a collection of declarations (e.g. values, datatypes, type classes).

- At the next lower level are expressions, which are the heart of Haskell programs. An expression denotes a value and has a static type.

- The bottom level is the lexical structure of Haskell. The lexical structure captures the concrete representation of Haskell programs in text files.
The computer is a calculator:
? 6 * 7
42

A more intellectually and challenging aspect of functional programming consists of building definitions. A list (sequence) of definitions is called “script”.

```
square :: Int -> Int
square x = x * x

smaller :: (Int, Int) -> Int
smaller (x,y)
  | x <= y   = x
  | otherwise = y
```
The purpose of a definition is to introduce a binding associating a given name with a given definition. A set of bindings is called an environment or context. Expressions are always evaluated in some context. This context must contain definitions for all occurrences of free (used) names used in the actual expression to be evaluated.
module FirstModule(square, smaller, greater) where

square :: Int -> Int
square x = x * x

smaller :: (Int,Int) -> Int
smaller (x,y) =
    | x <=y = x
    | otherwise = y

greater :: (Int,Int) -> Int
greater (x,y) =
    if x > y then x else y

Com S 541
Haskell Scripts

- Haskell scripts are collections of definitions.

- Definitions are expressed as equations between expressions and describe a mathematical function.

- Every definition has to be accompanied by a type signature.

- During a session, scripts can be loaded in order to introduce new definitions, which extent the current working environment.

- Definitions can contain references to other functions defined in preludes, libraries, or user-defined scripts.
The computer evaluates an expression by reducing it to the simplest equivalent form. The term *evaluation*, *simplification*, and *reduction* are used interchangeably to describe this process.

Consider the expression square (12 - 1), which has the following reduction tree:

```
  square(12 - 1)
       /     \
   square 11  (12 - 1) * (12 - 1)
           /     \            /     \
       11 * (12 - 1)  (12 - 1) * 11
           /      \     /     \
       11 * 11  121
```
Strict evaluation strategy reduces the leftmost-innermost expression (redex) first. Using this strategy, functions arguments are evaluated prior the function evaluation. This evaluation strategy is therefore also called \textit{call-by-value}, \textit{application order reduction}, or \textit{eager evaluation}.

\[
\text{square}( 12 - 1 )
\]
\[
= \{\text{definition of } - \}\ 
\text{square} \ 11
\]
\[
= \{\text{definition of square}\}\ 
11 * 11
\]
\[
= \{\text{definition of } * \}\ 
121
\]
Non-strict Evaluation

Non-strict evaluation strategy reduces the leftmost-outermost expression (redex) first. Using this strategy, functions arguments are only evaluated if needed, i.e., they are not evaluated prior function evaluation. This evaluation strategy is therefore also called *call-by-name*, *normal order reduction*, or *lazy evaluation*.

\[
\text{square}( 12 - 1 ) = \{\text{definition of square}\} \\
(12 - 1) * (12 - 1) = \{\text{definition of -}\} \\
(12 - 1) * 11 = \{\text{definition of -}\} \\
11 * 11 = \{\text{definition of *}\} \\
121
\]
An expression is said to be canonical, or in normal form, if it cannot be further reduced.

In the previous example (square (12 – 1)) we have seen that independent of the chosen evaluation strategy the final result was always the same – 121.

It is a characteristic feature of functional programming that if two different reduction sequences both terminate, then they lead to the same result. In other words, the meaning of an expression is its value and the task of the computer is simply to obtain it.

However, not every expression has a normal form: apply x = x x
apply apply → apply apply → apply apply → ... → apply apply
The Church-Rosser Property

- If an expression can be evaluated at all, it can be evaluated by consistently using normal-order evaluation. If an expression can be evaluated in several different orders (mixing normal-order and applicative-order evaluation), then all these evaluation orders yield the same result.

square (12-1):

- Applicative-order evaluation:
  \( \text{square } (12-1) \rightarrow \text{square } 11 \rightarrow 11 \times 11 \rightarrow 121 \)

- Normal-order evaluation:
  \( \text{square } (12-1) \rightarrow (12-1) \times (12-1) \rightarrow 11 \times (12-1) \rightarrow 11 \times 11 \rightarrow 121 \)

- Note, by means of lazy evaluation, we can both avoid having to calculate values before we need them, and also avoid having to calculate them more than once.
Infinity

For some expressions the process of reduction never stops and never produces any result.

three :: Int -> Int
three x = 3

infinity :: Int
infinity = infinity + 1

strict evaluation:
three infinity
= {definition of infinity}
three (infinity + 1)
= {definition of infinity}
three ((infinity + 1) + 1)
= ...

non-strict evaluation:
three infinity
= {definition of infinity}
3

Com S 541
infinity :: Int
infinity = infinity + 1

Such an expression do not denote well-defined values in the normal mathematical sense.

Another example is the operator \( \div \), which denotes numerical division, returning a floating-point number. However, the expression \( 1/0 \) does not denote a well-defined floating-point number.

Such expressions may cause the evaluator to respond with an error message (e.g. ‘divide by zero’), or go into an infinitely long sequence of calculations without producing any result.

In order to say that, without exception, every syntactically well-formed expression denotes a value, it is convenient to introduce a special symbol \( \bot \), pronounced “bottom”, to stand for the undefined value of a particular type.
Properties of Bottom

- The computer is not expected to produce the undefined value. Instead, the evaluator will produce an error message or run forever and may remain perpetually silent. The former situation is detectable while the latter is not (the evaluation simply takes more time depending on complexity class of the algorithm or the of function called – e.g. Ackermann function, while Turing computable, grows faster than any primitive recursive function).

- Therefore, $\bot$ is a special kind of value. In fact, every data type (e.g. Int, Float, tuple, function, etc.) has an additional element, namely $\bot$.

- If $f \bot = \bot$, then $f$ is said to be a strict function; otherwise it is non-strict. Thus, square is a strict function, while three is non-strict. Lazy evaluation allows non-strict functions to be defined, some other strategies do not.
Languages like Haskell support a number of styles for specifying which expressions should be evaluated for different cases of arguments:

fac :: Int -> Int

**Patterns:**

- fac 0 = 1
- fac n = n * fac (n-1)

Note that patterns are tested in their given order. If there is no matching pattern, the evaluation of the functions terminates with a run-time error.
Lists

- Lists are pairs of elements and lists of elements:
  - [] stands for the empty list
  - x:xs stands for the list with x as the head and xs as the rest of the list
  - [1,2,3] is syntactic sugar for 1:2:3:[]
  - [1..n] stands for [1,2,3,…,n]

- Lists can be constructed using patterns:
  - head (x:_) = x
  - len [] = 0
  - len (x:xs) = 1 + len xs
  - prod [] = 1
  - prod (x:xs) = x * prod xs
  - fac n = prod [1..n]
In languages like C and Pascal, the use of user-defined functions is very restricted. The types of the actual parameters have to match exactly the types of the formal parameter. Exceptions are only possible, if we switch off type checking, i.e. if we use type-casts.

Haskell is a strongly typed language, i.e. we can assign all values a (unique) type. However, it is possible to define functions, where the parameters of a function can be instantiated to different types in different circumstances. Such functions are called polymorph.

In order to use a polymorphic abstractions, Haskell provides type variables, and hence a type that is built using type variables is called a polymorphic type. We use \( \textbf{a} \), \( \textbf{b} \), \( \textbf{c} \) to denoted type variables:

\[
(\cdot) :: (\textbf{b} \to \textbf{c}) \to (\textbf{a} \to \textbf{b}) \to (\textbf{a} \to \textbf{c})
\]

\[
\text{map} :: (\textbf{a} \to \textbf{b}) \to \textbf{[a]} \to \textbf{[b]}
\]
What Is a Function?

- Functions are the most important kind of value in functional programming.

- It is important to note that we cannot display as function values. We can apply functions to arguments and display the results, when the result can be displayed (i.e. the result is not again a function).

- A function is a rule of correspondence that associates each element of a given type \( A \) with a unique element of a second type \( B \). The type \( A \) is called the source type, and \( B \) the target type of the function.

- We express this information by writing \( f :: A \rightarrow B \), which asserts that the type of \( f \) is \( A \rightarrow B \), i.e. \( f \) is a function from \( A \) to \( B \), like:
  - three :: Int -> Int
  - square :: Int -> Int
  - smaller :: (Int,Int) -> Int
Extensional and Intentional Views

- **Extensional view:**

  A (total) function $f: A \to B$ is a subset of $A \times B$ (i.e. a relation) such that:
  1. for each $a \in A$, there exists some $(a,b) \in f$ (i.e. $f(a)$ is defined), and
  2. if $(a,b_1) \in f$ and $(a,b_2) \in f$, then $b_1 = b_2$ (i.e. $f(a)$ is unique)

- **Intentional view:**

  A function $f: A \to B$ is an abstraction $\lambda x. e$, where $x$ is a variable name, and $e$ is an expression, such that when a value $a \in A$ is substituted for $x$ in $e$, then the expression (i.e. $f(a)$) evaluates to some (unique) value $b \in B$. 
Extensionality

- Two functions are equal, if they yield the same result for equal arguments, i.e. \( f = g \), if and only if \( f(x) = g(x) \) for all \( x \). This principle is called \textit{extensionality}, and states the correspondence between arguments and result.

\[
\begin{align*}
\text{double} :: \text{Int} &\rightarrow \text{Int} \\
\text{double} \ x &= x + x
\end{align*}
\[
\begin{align*}
\text{double'} :: \text{Int} &\rightarrow \text{Int} \\
\text{double'} \ x &= 2 \times x
\end{align*}
\]

- The two definitions describe different functions for obtaining the correspondence, one involving addition and the other involving multiplication, but \textit{double} and \textit{double'} define the same function value, and therefore it holds \( \text{double} = \text{double'} \). But one function may be more or less efficient that the other, but efficiency is an intentional property of a function.

- Extensionality means that we can prove \( f = g \) by proving \( f(x) = g(x) \) for all \( x \). Depending on the definitions of \( f \) and \( g \) we may be able to prove \( f = g \) directly.
Recursive functions can be less efficient than loops because of the high costs of function calls on most hardware. But a tail-recursive function calls itself only as its last operation, so the recursive call can be optimized away by a modern compiler.

A recursive function can be converted to a tail-recursive one by representing partial computations (i.e. state) as explicit function parameters:

```haskell
sfac :: Int -> Int -> Int
sfac s n = if n == 0
    then s
    else sfac (s*n) (n-1)
```

Tail-recursive functions are typically more efficient since no information about the previous call needs to be kept.
Multiple Recursion – Remembering State

- Naive recursion may result in unnecessary recalculations:
  
  \[
  \begin{align*}
  \text{fib} :: \text{Int} & \rightarrow \text{Int} \\
  \text{fib} 1 &= 1 \\
  \text{fib} 2 &= 1 \\
  \text{fib} (n+2) &= \text{fib} n + \text{fib} (n+1)
  \end{align*}
  \]

- Efficiency can be regained by explicitly passing calculated values:

  \[
  \begin{align*}
  \text{fib'} 1 &= 1 \\
  \text{fib'} n &= a \\
  \text{fib'} n &= a \\
  \text{fibPair} :: \text{Int} & \rightarrow (\text{Int},\text{Int}) \\
  \text{fibPair} 1 &= (1,0) \\
  \text{fibPair} (n+2) &= (a+b,a) \\
  \text{fibPair} (n+2) &= (a+b,a) \\
  \end{align*}
  \]

  where \(a,b) = \text{fibPair} (n+1)
Higher-order Functions

Higher-order functions treat other functions as first-class values that can be composed to produce new functions.

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

? map fac [1..5] -- fac implements the factorial function
[1,2,6,24,120]

Anonymous functions can be written as “lambda abstractions”:

? map (\x -> x * x) [1..10]
[1,4,9,16,25,36,49,64,81,100]

Note: map fac and map (\x -> x * x) are new functions that can be applied to lists.
Curried Functions

A Curried function takes its arguments one at a time, allowing it to be treated as a higher-order function.

\[
\text{plus } x \ y = x + y \quad \text{-- curried addition}
\]

? plus 1 2
3

\[
\text{inc } = \text{plus } 1 \quad \text{-- bind first argument of plus to 1}
\]

? inc 2
3

\[
\text{fac } = \text{sfac } 1 \quad \text{-- factorial binds first argument of}
\]

\[
\text{where sfac } s \ n \quad \text{-- a curried factorial function}
\]

\[
| n == 0 = s
| n >= 1 = \text{sfac } (s*n) \ (n-1)
\]

Curried functions are named after the logician H.B. Curry, who popularized them.
Currying

The following higher-order function takes a binary function as an argument and turn it into a curried function:

\[
\text{curry} :: ((a, b) \to c) \to (a \to b \to c)
\]

\[
\text{curry } f \ a \ b = f \ (a,b) \quad \text{-- take a binary function and curry it}
\]

\[
\text{plus} :: (\text{Int},\text{Int}) \to \text{Int}
\]

\[
\text{plus} \ (x,y) = x + y \quad \text{-- not a curried function}
\]

\[
\text{plusc} :: \text{Int} \to \text{Int} \to \text{Int}
\]

\[
\text{plusc} = \text{curry plus} \quad \text{-- make plusc the curried version of plus}
\]

\[
\text{inc} :: \text{Int} \to \text{Int}
\]

\[
\text{inc} = \text{plusc} \ 1 \quad \text{-- bind first argument of plus}
\]

It is left as an exercise to define a function \text{uncurry} that goes the other way and converts a curried function into a non-curried one.
The composition of two functions \( f \) and \( g \) is denoted by \( f \cdot g \) and is defined by the equation

\[
(\cdot) : (b \to c) \to (a \to b) \to (a \to c)
\]

\[(f \cdot g) \, x = f \, (g \, x)\]

In words, \( f \cdot g \) applied to \( x \) is defined to be the outcome of the first applying \( g \) to \( x \), and then applying \( f \) to the result.

Not every pair of functions can be composed since the types have to be matched up: we require that \( g \) has type \( a \to b \) for some types \( a \) and \( b \), and that \( f \) has type \( b \to c \) for some type \( c \).

Functional composition is an associate operation: \((f \cdot g) \cdot h = f \cdot (g \cdot h)\)
“Lazy”, or “normal-order” evaluation only evaluates expressions when they are actually needed. Clever implementation techniques allow replicated expressions to be shared, and thus avoid needless recalculations. So:

\[
\text{square } x = x \times x \\
\text{square } (12-1) \rightarrow (12-1) \times (12-1) \rightarrow 11 \times 11 \rightarrow 121
\]

Lazy evaluation allows some functions to be evaluated even if they are passed incorrect or non-terminating arguments:

\[
\text{ifTrue :: Bool} \rightarrow a \rightarrow a \rightarrow a \\
\text{ifTrue True } x \ y = x \\
\text{ifTrue False } x \ y = y
\]

?ifTrue True 1 (5/0)
1
Recall that a complete functional program is just a function from its input to its output. If \( f \) and \( g \) are such programs, then \( (f \cdot g) \) is a program which, when applied to its input, computes

\[
f(g \ input)
\]

The program \( g \) computes its output, which is used as the input to program \( f \).

This scheme might be implemented conventionally by storing the output from \( g \) in a temporary file (e.g. gzip, pipes-and-filters architecture). The problem with this is that the temporary file might occupy so much memory that it is impractical to glue the programs together in this way.

Functional languages provide a solution to this problem. The two programs \( f \) and \( g \) run in strict synchronization. \( G \) is only started once \( f \) tries to read some input, and only runs for long enough to deliver the output \( f \) is trying to read. Then \( g \) is suspended and \( f \) run until it tries to read another input.

This method is called “lazy evaluation” and provides a practical means to modularize a program. Moreover, in functional languages lazy evaluation is uniformly used for every function call.
Lazy lists are infinite data structures whose values are generated by need:

```haskell
from :: Int -> [Int]
from n = n : from (n+1)
take :: a -> [a] -> [a]
take 0 _ = []
take _ [] = []
take (n+1) (x:xs) = x : take n xs
```

```haskell
?take 5 (from 10)
[10,11,12,13,14]
```

Note: The lazy list (from n) has a special syntax: \([n..]\)

```haskell
? take 5 [20..] \rightarrow [20,21,22,23,24]
```

```haskell
fibs :: [Int]
fibs = 1 : 1 : fibgen 1 1
where fibgen a b = (a+b) : fibgen b (a+b)
```

```haskell
? take 10 fibs
[1,1,2,3,5,8,13,21,34,55]
```
Haskell provides an alternative notation, called a list comprehension, for describing computations involving map and filter:

\[ [x \times x \mid x \leftarrow [1..5], \text{odd } x] \]

\[ [1, 9, 25] \]

This reads: the list of squares of odd numbers in the range 1 to 5.

Formally, a list comprehension takes the form \([e \mid Q]\) where \(e\) is an expression and \(Q\) is a qualifier. A qualifier is a possible empty sequence of generators and guards.

A generator takes the form \(x \leftarrow xs\), where \(x\) is a variable or tuple of variables, and \(xs\) is a list-valued expression. A guard is a boolean-valued expression.

\begin{align*}
\text{pairs} &= [(i,j) \mid i \leftarrow [1..2], j \leftarrow [1..3]] \\
\text{?} &\ [i+j \mid (i,j) \leftarrow \text{pairs}] \\
\text{[2,3,4,3,4,5]} \\
\end{align*}
Consider the algorithm of Eratosthenes to find all prime numbers:
1. Create a list that contains all natural numbers,
2. Mark the first unmarked number,
3. Delete all multiples of the last marked number,

This algorithm yields a list of all prime numbers, but the algorithm does not terminate since there exists an infinite number of prime numbers. In general, we are however only interested in a finite sequence starting with 2. This gives us an intuition that can be implemented best with a non-strict (lazy) language.

Functional programs can often be derived in a top-down fashion. Therefore, we decompose the problem in several sub-problems that provide a finer-grained abstractions.
The Sieve of Eratosthenes

primes :: [Int]
primes = dropMultiples [2..] -- generate a list of all natural numbers

-- mark first unmarked number, drop multiples of marked number
-- incorporate step 4: continue at step 2 \rightarrow recursive call of dropMultiples

dropMultiples :: [Int] -> [Int]
dropMultiples (x:xs) = x : dropMultiples (primeFilter (\y -> (y `mod` x) /= 0) xs)

-- delete all multiples

primeFilter :: (a -> Bool) -> [a] -> [a]
primeFilter p [] = []
primeFilter p (x:xs)
    | p x = x : primeFilter p xs
    | otherwise = primeFilter p xs

? take 100 primes
[2,3,5,7,11,13,...,523,541]
One advantage of the non-strict nature of Haskell is that data constructors are non-strict too. Non-strict constructors permit the definition of (conceptually) infinite data structures.

```haskell
ones :: [Int]
one = 1 : ones
```

We have

```haskell
ones = 1 : ones = 1 : 1 : ones = 1 : 1 : 1 : ones = ...
```

The representation on ones as a graph has, however, a particularly interesting property, since it involves a cyclic structure. Therefore, the entire infinite list can be represented with a fixed amount of space:
Lazy Patterns

repeat \( x = x : \text{repeat } x \)

The definition of repeat is equivalent to the former definition of ones, but does \emph{not} create a cyclic structure. Therefore, the list ones grows longer with each evaluation:

\[
\text{repeat } 1 = 1 : 1 : 1 : 1 : 1 : \text{repeat } 1
\]

However, if we define

\[
\text{repeat } x = \text{let } xs = x : xs \text{ in } xs
\]

then the definition of ones in terms of repeat will produce again a cyclic structure.

The binding created by the let expression is mutually recursive, and the pattern binding is treated as a lazy pattern, written \( \sim \text{pat} \). A lazy pattern is an irrefutable pattern, i.e. pattern matching always succeeds. In the Haskell report the following translation for the above let expression is given:

\[
\text{let } p = e_1 \text{ in } e_0 = \text{let } p = \text{fix } ( \lambda p \rightarrow e_1 ) \text{ in } e_0
\]

where \text{fix} is the least fixed point operator.
repeat x = x : repeat x

Therefore, we get: 1 : 1 : 1 : ... : 1 : repeat 1

However, with
repeat x = xs
where xs = x : xs